Concise Promela Reference

by Rob Gerth, June 1997

This is a quick reference for things that can be found in the **Spin** man pages. It is less cursory on matters the discussion of which is scattered through the various **Spin** documentation files or only found in papers. In particular, sections **Execution** and **Analysis** are more descriptive. This reference is based on **Spin** version 2.9.7. For a slightly more detailed description see the Basic Spin Manual. A full description can be found in the Spin book.

Introduction

Spin is a tool for analyzing the logical consistency of distributed systems, specifically of data communication protocols. The system is described in a modeling language called **Promela** (Process or Protocol Meta Language). The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous (i.e., rendez-vous), or asynchronous (i.e., buffered). **Xspin** is a graphical front-end to drive **Spin** (written in Tcl/Tk).

Given a model system specified in **Promela**, **Spin** can perform random or interactive simulations of the system's execution or it can generate a **C** program that performs a fast exhaustive verification of the system state space. During simulations and verifications **Spin** checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The verifier can also be used to prove the correctness of system invariants and it can find non-progress execution cycles. Finally, it supports the verification of linear time temporal constraints; either with **Promela** never-claims or by directly formulating the constraints in temporal logic.

The verifier is setup to be fast and to use a minimal amount of memory. The exhaustive verifications performed by **Spin** are conclusive. They establish with certainty whether or not a system's behavior is error-free. Very large verification runs, that can ordinarily not be performed with automated techniques, can be done in **Spin** with a ``bit state space'' technique. With this method the state space is collapsed to a few bits per system state stored. Although this technique doesn't guarantee certainty, the coverage is better, and often much better, than that obtained with traditional random simulation.

Promela

Spin models consist of 3 types of objects: *processes*, message *channels*, and *variables*. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run.

The syntax of **Promela** is **C**-like. Below, our tendency is to use examples for those parts that are similar to **C**, but we attempt to be more precise elsewhere.

Lexical conventions

There are five classes of tokens: identifiers, keywords, constants, operators and statement separators. Blanks, tabs, newlines, formfeeds and comments serve only to separate tokens. If more than one interpretation is possible, a token is taken to be the longest string of characters that can constitute a token.

Comments

A comment starts with `/*` and terminates with `*/`. Comments may not be nested.

Identifiers

An identifier is a single letter, period, or underscore followed by zero or more letters, digits, periods, or underscores.

Keywords

The following identifiers are reserved for use as keywords.

```
active          assert          atomic          bit
bool            break           byte            chan
d_step          D_proctype      do              else
empty           enabled         fi              full
goto            hidden          if              init
int             len             mtype           nempty
never           nfull           od              of
pc_value        printf          priority        proctype
provided        run             short           skip
timeout         typedef         unless          unsigned
xr              xs
```

Labels

A label is an identifier followed by a colon (`:`). Any statement can be labeled. Labels that start with one of the sequences `end`, `progress` or `accept` have a special meaning, discussed in Section **Analysis** below.

Constants

A constant is a sequence of digits representing a decimal integer. There are no floating point numbers in **Promela**. Symbolic names for constants can be defined in two ways. The first method is to use a **C**-style macro definition

```
#define NAME    5
```

The second method is to use the keyword `mtype`; see symconst.

Expressions

The following operators and functions can be used to build expressions:

```
+       -       *       /       %       >
>=      <       <=      ==      !=      !
&       ||      &&      |       ~       >>
<<      ^       ++      --
len()   empty() nempty()        nfull() full()
run     eval()  enabled()       pc_value()
```

Most operators are binary. The logical negation (`!`) and the minus (`-`) operator can be both unary and binary, depending on context. The `++` and `--` operators are unary suffix operators, like they are defined in **C**. The exclusive-or is `^`. The functions on the one-but-last line are unary and apply only to message channels (see MsgChan); `len` measures the number of messages an existing channel holds; the other channel operators have the expected meaning and are introduced to assist partial order reductions; see pored. These functions cannot be used in larger expressions. E.g., `!empty(q)` will result in a syntax error and `nempty(q)` should be used instead. The unary functions on the last line are special. The first operator is used for process instantiation (see Processes below). When executed it yields the instantiation number of the instance it created. The second one is discussed in ChanOps and the others in never. The operators on the first four lines are defined as in **C**. **Promela** follows the **C**-convention that the boolean false-condition corresponds with the value 0; any non-zero value denotes truth.

Conditional expressions

```
(expr1 -> expr2 : expr3)
```

has the value of `expr3` if `expr1` evaluates to zero, and the value of `expr2` otherwise. Note that `->` is required here and cannot be replaced by a `;`.

Declarations

Processes, channels, variables, etc. must be declared before they can be used. Variables and channels can be declared either locally, within a process, or globally. A process can only be declared globally in a `proctype` declaration. Local declarations may appear anywhere in a process body.

Variables

A variable declaration is started by a keyword indicating the basic data type of the variable, `bit`, `bool`, `byte`, `short` or `int`, followed by one or more identifiers, optionally followed by an initializer:

```
byte name1, name2 = 4, name3
```

An initializer, if specified, must be a constant. By default variables are initialized to zero. The bit-widths of these basic types are, respectively, 1, 1, 8, 16 and 32. The last two are *signed* quantities; the first three are unsigned. A variable can have a user-defined type as well; see structs.

Arrays

An array of variables is declared, e.g., by

```
int name[4]
```

An array can have a *single* constant as an initializer, initializing all array elements. Array indices start at 0, as in **C**; hence, the largest index would be 3 in this case.

Symbolic constants

Symbolic constants can be declared by

```
mtype = {OK, READY, ACK}
```

and variables of this type by

```
mtype Status = OK;
```

Only one `mtype`-definition is allowed which must be global and at most 256 symbolic constants can be declared; an `mtype` variable is 8 bits wide.

The advantage of `mtypes` over `#defines` is that the former type of symbolic constants is recognized by **Spin** and during simulations the symbolic names are used instead of the values they represent.

Message channels

Message channels are declared, e.g., by

```
chan Transfer  = [2] of {mtype, bit, short, chan};
chan Device[3] = [0] of {byte};
chan Channel;
```

Here, `Transfer` can store up to 2 messages in the channel; the message type is indicated between the braces (in this case each message consists of 4 parts). `Device` is an *array* of channels; each channel is synchronous, i.e., sends and receives must synchronize as no messages can be stored. Finally, `Channel` is an uninitialized channel that can be used only after an initialized channel has been assigned to it.

Note that an object of type `chan` can be part of a message declaration in another channel.

Exclusive receive (xr) and send (xs) on channels

```
xr Transfer;
xs Channel;
```

in some process, declares that this process is the *only* one to receive messages on `Transfer` and the *only* one to send on `Channel`.

It is a run-time error if during analysis it turns out that some other process receives from `Transfer` or sends to `Channel`.

See Reductions for more detail.

Structures

User-defined data types are supported through `typedef` definitions,

```
typedef Msg {
        byte a[3], b;
        chan p
}
```

that can be used in variable declarations and, more generally, wherever a type definition is needed:

```
Msg foo;
chan stream =[0] of {mtype,Msg}
```

Elements of structures are accessed as in C; e.g.,

```
foo.a[1]
```

Hidden variables

```
hidden int foo
```

A `hidden` variable is not part of the system state (see **`Execution`**) and its value is always undefined, although it can be assigned to. It is useful when a `scratch' variable is needed, e.g., to flush the values in a channel buffer, because it does not add to the size of the state-vector during analysis and thus reduces the memory requirements for analysis; cf. MemTim.

Only global declarations can be thus qualified; elements of a structure are considered local. Bit, bool and channel variables cannot be hidden.

There is a pre-declared hidden variable, `_`, which can be assigned to in any context. Its (implicit) type is `int` so that it can be used to assign `bit`, `bool`, `byte`, `mtype`, `short` and `int` values to.

`Processes`

A basic process declaration has the form

```
proctype pname( chan In, Out; byte id )
{ statements }
```

Such a process is instantiated by a run-operation:

```
run pname(Transfer, Device[0], 0)
```

that first assigns the actual parameters to the formal ones and then executes the `statements` in the body. Each process instance has a unique, positive instantiation number, which is yielded by the run-operator (and by `pid`); see specvar. A process-instance remains active until the process' body terminates (if ever).

Processes cannot have arrays as (part of) a formal parameter type, but structures are allowed.

Process declarations can be augmented in various ways. The most general form is

```
active [N] proctype pname(...) provided (E) priority M
```

The `active` modifier causes `N` instances of the proctype to be active in the initial system state, where `N` is a constant. If `[N]` is absent, only one instance is activated. Formal parameters of instances activated through the `active` modifier are initialized to 0; i.e. actual parameters can only be passed using `run`-statements.

A proctype can have an *enabling* condition `E` associated with it, which is a general side-effect free expression that may contain constants, global variables, the predefined variables `timeout` and `pid`, but not other local variables or parameters, and no remote references. Enabling conditions are evaluated once, in the initial state.

For use during random simulations, a process instance can run with a priority `M`, a constant >=1. Such a process is `M` times as likely to be scheduled than a default (priority 1) process. Execution priorities can be used in a run-statement as well:

```
run pname(...) priority M
```

A process instantiated with a `run` statement always gets the priority that is explicitly or implicitly specified there (the default is 1).

Note that priorities have no effect during analysis.

Deterministic processes

```
D_proctype pname( chan In, Out; byte id )
{ statements }
```

declares that any instance of `pname` is deterministic. It has no other effect, then causing an error during analysis if some instance is not.

Note that determinism is a dynamic property. E.g., if `pname` has in its body the statement

```
if
:: In?v  -> ...
:: Out!e -> ...
fi
```

then non-determinism is flagged only if during some computation, there is an instance of `pname` for which the receive and send on the actual channels bound to `In` and `Out` are simultaneously enabled.

Init process

```
init { statements }
```

This process, if present, is instantiated once, and is often used to prepare the true initial state of a system by initializing variables and running the appropriate process-instances.

Never claim

```
never { statements }
```

is a special type of process that, if present, is instantiated once. As explained further in tempclaim, never claims are used to detect behaviors that are considered undesirable or illegal.

The individual statements in `statements` are interpreted as conditions (see Exec and ExprStat) and, therefore, should not have side-effects (although this will cause warnings rather than syntax-errors).

Statements that can have side-effect are assignments, auto-increment and decrement operations, communications, run and assert statements.

Never claims can use three additional functions:

- `enabled(pid)` This boolean function returns true if the process with instantiation number denoted by `pid` is able to perform an operation. This function can only be used in systems that do not admit *synchronous* communication
- `pc_value(pid)` This function returns the number of the state that the process with instantiation number denoted by `pid` is currently in. (With state-numbers as given by the `-d` runtime option.)
- `_np` This predicate is true if the system is in a non-progress state and is false otherwise. It is introduced to enable more efficient searches for non-progress cycles. See progress and SearchNPC.

Special variables

```
_pid, _last
```

`_pid` is a predefined local variable that holds the instantiation number of the process' instance. `_last` is a predefined global variable that holds the instantiation number of the process instance that performed the last step in the current execution sequence. Initially, `_last` is zero.

Remote references

The expression

```
procname[pid]@label
```

is true precisely if the process with instantiation number `pid` of proctype `procname` is currently at the statement labeled with `label`.

`Statements`

The following can be used as statements:

```
assert          assignment      atomic          break
declaration     d_step          else            expression
goto            receive         selection       skip
repetition      send            timeout         unless
```

In particular, note that expressions can be used as statements as well.

Statements are separated by either a semi-colon (`;`) or, equivalently, an arrow (`->`). The arrow is sometimes used to indicate a causal relation between successive statements and also in selection and repetition statements to separate the guard from the statements it is guarding; see select and repet.

Statements that have no smaller statements as a constituent, are called *basic*. E.g., an assignment is a basic statement whereas a selection is not.

Executability

The execution of a statement is conditional on its *enabledness* (or ``executability''). Statements are either enabled or *blocked*. Of the above listed statements, assignments, declarations, `assert`, `skip`, `goto` and `break` are always enabled. If a statement is blocked, execution at that point halts until the statement becomes enabled.

Assert

```
assert( expression )
```

aborts the program if the `expression` returns a zero result; otherwise it is just passed.

Atomic

```
atomic { statements }
```

attempts to execute the `statements` in one indivisible step; i.e., without interleaved execution of other processes. An atomic statement is enabled if its first statement is.

Making local computations atomic can effect important reductions of the complexity of the verification system; cf. MemAtD.

During its execution, control can only be transferred outside the scope of an atomic statement by an explicit `goto` or at a point where a statement within its scope becomes blocked. If this statement subsequently becomes enabled again, execution may continue at that point.

There is no constraint on what may occur inside the scope. In particular, it is possible to jump to any (labeled) location within the scope of an `atomic`.

The body of a process instance activated by a `run`-statement is considered to be outside the scope of the atomic statement performing the activation.

Break

See Repetition.

Goto

```
goto label
```

transfers control to the statement labeled by `label` which has to occur in the same procedure as the goto.

Deterministic step

```
d_step { statements }
```

has the same effect as `atomic` but is even more efficient; cf. MemAtD. However, `statements` within its scope must be completely deterministic; they may not jump to labels outside the `d_step`'s scope; there may be no jumps from the outside to labeled statements within the scope; remote references (see RmtRef) are disallowed. Finally, statements other than the first may not block. A `d_step` is enabled precisely if its first statement is. **`Promela`** considers the location in front of the `d_step` to be within its scope and the location just after its last statement to be outside. Hence, to achieve the intended effect of the following *incorrect* code-fragment:

```
        goto label;
        ...
label:
        d_step {
                ...
                do
                ...
                        break
                ...
                od
        }
```

use this code instead:

```
        goto label;
        ...
label: skip;
        d_step {
                ...
                do
                ...
                        break
                ...
        od; skip
        }
```

Else

See select.

Expressions

Any expression that does not use auto-increment or decrement operations (`++` or `--`) can be used as a statement. In that case, it is enabled precisely if it evaluates to a non-zero value. An enabled expression is just passed, without affecting the state.

For example, instead of writing a busy wait loop

```
while (a != b) skip
```

the same effect is achieved in **`Promela`** by the statement

```
(a == b)
```

Channel operations

```
q!var1,const,var2,...  or, equivalently  q!var1(const,var2,...)
q?var1,const,var2,...  or, equivalently  q?var1(const,var2,...)
```

Are the standard channel operations; the first is a send statement, the second a receive. Here, `q` denotes a channel and the list `var1,const,var2,...` should be compatible with the channel's message type. For a send or receive to be enabled, `q` has to be initialized. Furthermore, if the channel is buffered, a send is enabled if the buffer is not full; a receive is enabled if the buffer is non-empty. On an unbuffered channel, a send (receive) is enabled only if there is a corresponding receive (send) that can be executed simultaneously. A receive statement executes by reading the oldest message on the channel; if the channel is unbuffered, it reads the message of the simultaneously executing send statement. A send statement executes by putting its message in the buffer (if there is one). Note that a channel operation on an unbuffered channel can only execute if a matching operation executes simultaneously.

Constants in the list of a receive, constrain its enabledness by forcing the corresponding values in the oldest message (or matching send) to be the same; if not, the receive is blocked.

It is possible to use a local or global variable to likewise constrain a channel operation's enabledness:

```
q?var1,eval(var2),var3
```

blocks in case a matching send's 2nd value does not equal the *value* of `var2`. Note that the value of `var2` is not changed.

For *buffered* channels, there are additional operations:

- `q?[var1,const,var2]` returns true (i.e. a non-zero value) precisely if the corresponding receive operation would be enabled.
- `q?<var1,const,var2>` like a standard receive operation except that the message is *not* removed from the buffer.
- `q!!var1,const,var` *sorted send*; it inserts its message into the buffer immediately ahead of (so that it will be younger than) the oldest message that succeeds it in numerical, lexicographic order.
- `q??var1,const,var` *random receive*; it executes if there is *some* message in the buffer for which it is enabled and then retrieves the oldest such message.
- `q??[var1,const,var2]` returns true (i.e. a non-zero value) precisely if the corresponding receive operation would be enabled.
- `q??<var1,const,var2>` like a standard random receive operation except that the message is *not* removed from the buffer.

The behavior of buffered channels can be influenced by the **`Spin`** command line switch `-m`: in that case, send actions on a channel do not block if the channnel's buffer is full; instead, messages send when the buffer is full are lost.

Selection

```
if
:: statements
...
:: statements
fi
```

Selects one among its *options* (each of them starts with `::`) and executes it. An option can be selected if its *first* statement (the *guard*) is enabled. A selection blocks until there is at least one selectable branch. If more than one option is selectable, one will be selected at random. The special guard `else` can be used (once) in selection and repetition statements and is enabled precisely if all other guards are blocked. It may not be used if a send or receive statement is used as guard.

Repetition

```
do
:: statements
...
:: statements
od
```

Similar to a selection, except that the statement is executed repeatedly, until control is explicitly transferred to outside the statement by a `goto` or `break`. A `break` will terminate the innermost repetition statement in which it is executed and cannot be used outside a repetition.

Skip

Has no effect and is mainly used to satisfy syntactic requirements.

Timeout

A `timeout` statement becomes enabled precisely when every other statement in the system is blocked. It has no effect when executed.

Unless

```
{ statements-1 } unless { statements-2 }
```

Starts execution in `statements-1`. Before each statement in `statements-1` (including the first one) is executed, enabledness of `statements-2` is checked and if it is, execution of `statements-1` is aborted and control transfers to `statements-2`; control remains in `statements-1`, otherwise. If `statements-1` terminates, `statements-2` is ignored.

In an `unless`, a `d_step`, in contrast with an `atomic`, is considered indivisible; i.e., enabledness of `statements-2` is not checked if control in `statements-1` resides within the scope of a `d_step`.

Executing a Promela system

A *system state* of a **`Promela`** system comprises the values of the global variables, the contents of the channel buffers and for each process instance, the values of its location counter, its local variables and local channel buffers. If a never claim is present, the state also includes the value of the claim's location counter.

Initially, all global (non-initialized) variables contain the value 0 and the channel buffers are empty. The existing process instances are the ones created through the `active` modifier on the `proctype` declarations together with the `init` and `never` processes, if present. The location counters of these instances are at their first statements.

A system state of a **`Promela`** system uniquely identifies the enabled statements in every (active) process instance. An *execution step* in a system *without* never claim, proceeds by arbitrarily selecting one of the enabled statements and then executing that statement, thus arriving in a new system state. So, process instances execute asynchronously. Note, however, that in case the selected statement was an unbuffered send (receive), a corresponding receive (send) statement was executed simultaneously.

If the system does have a never claim, an *execution step* is a combined, synchronized step: executing an (enabled) statement from the never claim together with a step from the rest of the system, as above. Thus, a never claim blocks execution of the system, in those situations in which the never claim has no enabled statements (in the current state); also see tempclaim.

The *execution sequences*, or *behaviors*, of a **Promela** system are the maximal (possibly infinite) sequences of execution steps, in which the first step is taken from the initial system state and each successive step from the state produced by the preceding step. In particular, notice that **Promela** allows starvation of processes during an execution. This behavior can be influenced by the analyzer command line switch `-f` (the weak fairness option). If used, process starvation cannot occur; i.e., execution sequences along which some process instance eventually does not make a move anymore although the instance remains continuously enabled are no longer possible.

Atomic statements and d_steps

Atomic statements and `d_steps` constrain execution elsewhere in the system, as they execute (in principle) as a single basic statement. Accordingly, in the presence of a never claim, an execution step of the system combines a *single* step in the never claim with an execution *sequence* of the atomic statement or `d_step` (defined as if there were no never claim). Note that in the case of an atomic statement, such an execution sequence may end in the middle of it and the above holds for subsequent steps within the atomic statement as well.

Analysis

A **Promela** system can be *exhaustively* (but efficiently) analyzed for *correctness violations*: i.e., for the existence of execution sequences that abort through an `assert`; that end in an invalid *end*-state; that avoid cycling through certain desirable, so-called *progress* states or that cycle through undesirable, so-called *acceptance* states; and for executions satisfying general (linear time) temporal properties (or claims) defined through *never-claims*.

End-states

Valid end-states are those system states in which every process instance and the init process has either reached the end of its defining program body or is blocked at a statement that is labeled with a label that starts with the prefix `end`. All other states are *invalid* end-states.

Strictly speaking, valid end-states should also require channels to be empty. Conformance with this stricter condition is obtained through **Spin**'s `-q` option.

When checking for *state properties*, the verifier will complain if there is an execution that terminates in an invalid end-state.

Progress states

A progress state is any system state in which some process instance is at a statement labeled with a label that starts with the prefix `progress`; a *progress label*.

When checking for *non-progress cycles*, the verifier will complain if there is an execution that does not visit infinitely often a progress state.

Acceptance states

An acceptance state is any system state in which some process instance is at a statement labeled with a label that starts with the prefix `accept`; an *acceptance label*.

When checking for *acceptance cycles*, the verifier will complain if there is an execution that visits infinitely often an acceptance state.

Temporal claims

Temporal claims are defined by **Promela** `never` claims (see never) and are used to detect behaviors that are considered undesirable or illegal.

When checking for state properties, the verifier will complain if there is an execution that ends in a state in which the never claim has terminated; i.e., has reached the closing } of its body. When checking for acceptance cycles, the verifier will complain if there is an execution that visits infinitely often an acceptance state. Thus, a temporal claim can detect illegal infinite (hence cyclic) behavior by labeling some statements in the never claim with an acceptance label.

In such situations the never claim is said to be *matched*. In the absence of acceptance labels, no cyclic behavior can be matched by a temporal claim. Also, to check a cyclic temporal claim, acceptance labels should only occur within the claim and nowhere else in the **Promela** system.

A never claim is intended to monitor every execution step in the rest of the system for illegal behavior and for this reason it executes in lock-step. Such illegal behavior is detected if the never claim matches along a computation. If a claim blocks (because no statement in its body is enabled) but it is not at its closing }, then there is no need to explore this computation any further because it cannot lead to a violation; whence a blocked claim terminates execution in the rest of the system so that other executions can be explored.

Note that if a never claim exhibits non-determinism then, according to the definition of execution sequence, the different ways in which this non-determinism is resolved result in different executions.

Example claim

Let `p` and `q` be two boolean expressions and consider the property that

> ``*along every computation, each system state in which `p` is true (a `p`-state) is eventually followed by a `q`-state*''

The following never claim verifies whether the property holds; i.e., it will detect any violation of the property:

```
never {
        do
        :: p -> break
        :: true         /* or equivalently: (p || !p) */
        od;
accept:
        do
        :: !q
        od
}
```

The first repetition terminates only in `p`-states. Such a state should eventually be followed by a `q`-state. The second repetition (hence the never claim) cannot terminate, so the never claim either eventually blocks because the computation sequence reaches a `q`-state or matches because the (infinite) computation cycles through an acceptance state. The latter occurs precisely if there are no subsequent `q`-states. Because the analyzer guarantees an exhaustive search for computations along which the never claim is matched, a computation violating the property is guaranteed to be detected (if there is one).

Note that replacing `skip` by `else` would check only the *first* occurrence of a `p`-state along computations. Stated differently, by giving the first repetition the non-deterministic choice in `p`-states to either break or continue and by relying on the exhaustive search mechanism of the analyzer, every occurrence of a `p`-state is checked. In fact, one can show that it is impossible to check this property without using non-determinism as above.

Memory and Time requirements

For this, a bit more must be said about the actual analysis process. The basic process is recursive (though its implementation is not) and starts in the initial system state. In each system state, the list of enabled actions is determined. Then each action in this list is selected, it is executed and the procedure is recursively applied to the new system state. If a never claim is present, the list becomes a list of action-*pairs*: one from the never claim together with an action (or execution sequence if an `atomic` or `d_step` is involved) from some active process instance. The recursion backtracks from any system state that has already been visited or in which there is no enabled action (pair) that has not already been selected in this state. Thus, the basic process is a depth-first generation of all (reachable) system states. In reality the process is more complex because the absence of correctness violations need be established as well.

As to the memory demands when checking for state properties, the visited system states need to be stored in order to decide when to backtrack, as well as the partial computation that is being extended (the `stack' in terms of depth-first generation) in order to know where to backtrack to. This, together with the number of bytes needed to represent a system state, i.e., the size of the *state vector*, determines the amount of memory that is needed to complete an analysis. In particular, the set of already visited system states must be kept in main memory as this set is accessed in an, essentially, random way during analysis. This is the reason why the size of the system that can be analyzed is directly determined by the amount of physical RAM.

If analysis has to establish the presence (progress) or absence (acceptance) of cyclic computations, both the number of stored states and the size of the `stack' at worst doubles when compared to the case of checking state properties. Note that checking a never claim corresponds to the analysis of a property in one of these three classes. Also note that adding a non-deterministic never claim to a system will in itself increase the state space so that it is quite possible that the memory demands increase more than twofold when searching for behavior matching a never-claim that has an acceptance state.

Analysis time is proportional to the number of visited configurations.

Atomic statements and d_steps

An atomic statement or `d_step` causes the statements in its scope to execute, in principle, as a single basic statement. Accordingly, the number of system states is reduced, whence the memory needed to record the visited (global) states, because no process instance can change state other than the one in which control resides within the atomic statement or `d_step`. In addition, system states in which control resides *inside* some `d_step` need not be recorded at all; not even on the stack.

Partial order reduction

This is an optimized search technique that decreases the memory and time needed for an analysis by an order of magnitude in most cases; sometimes substantially more. It will never increase the necessary memory and, at worst, only slightly increase the analysis time. The reduction method is invoked by default and the analysis results are guaranteed to be the same as those obtained without reduction. Generally speaking, partial order reduction works during analysis by deciding in each system state where there is non-determinism, whether the property that is being verified depends on the order in which execution steps are taken. If it is independent of this order, the analyzer fixes an ordering and ignores executions in which these steps are taken in a different order. Thus, the effect is as if pieces of **Promela** code had been enclosed in atomic statements, except that they are *dynamically* placed. I.e., the execution order of syntactically the same two statements can influence the property in one system state, whereas the property can be independent of the ordering in another.

Partial order reduction as implemented in **Spin** has the most effect on loosely coupled systems that interact through shared, global variables and/or *buffered* channels. If a system has no shared variables and uses only unbuffered communication channels, there will be no reduction in memory or time.

Stutter closedness

The only requirement for using partial order reduction is that a **Promela** `never` claim must be *stutter-closed*. This means that the property expressed by the claim must not depend on the number of execution steps that it remains true or false. For example, the two never claims in the example are stutter-closed, whereas the claim

```
        never { p; p }
```

is not, since it depends on the expression `p` remaining true for at least one execution step (and it has to be true in the initial system state).

Any property that can be expressed in linear temporal logic without using the `next-state' (X) operator is guaranteed to be stutter-closed. In fact, the above never claim would correspond to the temporal logic formula: `p && X p`.

In fact, independently of whether partial order reduction is used or not, it is probably a good idea to only verify stutter closed never claims.

Searching for non-progress cycles

Non-progress cycles can be detected using the runtime `-l` option. This will cause a search for acceptance cycles with the following never claim compiled in:

```
never {
        /* non-progress: <>[] np_ */
        do
        :: skip
        :: np_ -> break
        od;
accept:
        do
        :: np_
```

```
        od
}
```

This never claim matches along behaviors on which from some point onwards, the predicate `np_` remains `true`; i.e., only non-progress states are visited.

Note that for more complex non-progress properties, the user has to supply her own never claim and it is here that the `np_`-predicate is of use.

Increasing the amount of reduction

If a system uses buffered communications, the effectiveness of partial order reduction can be enhanced by declaring `proctypes` to have exclusive access to channels (`xr` and `xs`) whenever this is the case. However, doing so constrains the way a channel can be accessed:

```
        If you declare   Other processes may only do
        xr q             send actions on q
                         nfull(q)
        xs q             receive actions on q
                         nempty(q)
```

Any other type of access will give an error during analysis. In particular the functions `full()` and `empty()`, which are included for reasons of symmetry.

Bitstate hashing

Bitstate hashing is used in cases where memory is insufficient to perform an exhaustive analysis (even with partial order reduction). It is compatible with partial order reduction. Bit-state hashing applies to larger models but (or, rather, because) it does not guarantee an exhaustive analysis. On the other hand, when set-off against classical random simulation techniques, it is always better to use bit-state hashing because the coverage (i.e., the number of visited stated relative to the number of actual states) is never worse, and usually much better, than that achieved with random simulation. Using this technique, the analyzer will never incorrectly flag errors; however, it may miss errors.

Bit-state hashing works by allowing more states to be stored in the memory, `M`, set aside for recording visited states; the memory needed to store the stack is not reduced. It achieves this reduction by viewing every bit in `M` as a bucket in a *hash table*. Each system state is hashed onto a fixed number of buckets (i.e., 2), but only the fact that a bucket has become filled is recorded. In other words, there is *no collision detection* and if two system states hash onto the same buckets, they are deemed to be the same states. It is this feature that allows many more states to be stored. E.g., if the state vector of a system is 100 bytes then bit-state hashing, in the ideal case of a perfect hashing function, allows upto 800 times as many states to be stored and visited as would be possible with an exhaustive analysis using the same amount of memory.

Reality is more complex. Hash functions are not perfect, collisions will occur and get more common the fuller the hash table becomes and each collision potentially causes a part of the state space to remain unexplored.

The analyzer gives an indication of the quality of a bit-state search in the form of a *hash factor*: the fraction of buckets that were filled. I.e., a hash-factor of 100 means that, on average, at most 1 out of every 100 bits in `M` was set. This is taken as indicating a good quality search. **`Spin`** reports in this case ``expected coverage: >= 99.9% on avg.''. For a hash-factor between 10 and 100, **`Spin`** gives an expected coverage of 98% on average. These are heuristic estimates, and the variance can be large. E.g., experiments indicate that `98% coverage on average' can mean as low as 84% during an actual analyzer run. On the other hand, such coverage still is much superior over what is achieved by a standard random simulation.

Spin Online References
Promela Manual Index
Promela Grammar

Spin HomePage